LBL
August 1996
preliminary

Evaluations of Object Oriented Databases for Storage and Retrieval of BaBar Conditions Information

J. Ohnemus
Lawrence Berkeley Laboratory
Berkeley, California 94720, USA

Abstract

Object oriented database management systems have been evaluated as a possible means of storing and retrieving conditions information (calibration and geometry data) for the BaBar experiment. This paper describes the main features of object oriented database management systems and presents the results of benchmark performance tests on two commercial systems, Objectstore and Objectivity. Recommendations for the database system best suited for use in the BaBar experiment are given.

1 Introduction

Object oriented database management systems (OODBMS) provide a convenient and powerful means of storing and retrieving large amounts of complex data, thus they are potentially well suited for use in a high energy physics experiment. The main advantages of using a commercial database system is that it eliminates the lengthy and costly process of developing and testing an in-house database system. Two commercial object oriented database management systems have been evaluated as potential candidates for storing conditions information for the BaBar experiment. (Conditions information is a generic name for calibration and geometry data.) The first system is Objectstore, which is made by Object Design, Inc. and the second is Objectivity, which is produced by Objectivity, Inc.. This paper reports the results of performance evaluations of these two systems. The purpose of these evaluations is to make a recommendation for the choice of the database system to use in the BaBar experiment.

The remainder of this paper is organized as follows. The main features of object oriented database management systems are described in Section 2. Features which are specific to Objectstore and Objectivity are discussed in Sections 3 and 4, respectively. The results of benchmark tests on the two database systems are presented in Section 5. Section 6 contains a summary of the database benchmark results along with recommendations for the database system best suited to the needs of the BaBar experiment.

2 Object Oriented Databases

Objectstore and Objectivity both provide features commonly found in an object oriented database system, however, their nomenclature and style for implementing the features are different. This section begins by defining some of the basic concepts associated with databases and is followed by a description of the main features common to all object oriented database systems.

The Objectstore and Objectivity database systems are both written in **C++** and will operate on most common workstations running most common operating systems. In addition, it is also possible to compile application code with most **C++** compilers. The benchmark results presented in this paper have been obtained using the latest versions of these database systems, namely, Objectstore 4.0 and Objectivity 3.8. Furthermore, the work was done on a Sun Sparc 5 workstation (64 MB of RAM) running the Solaris 2.4 operating system and using the Sun **C++** 4.0 compiler.

A database management system provides persistence, which is the long-term storage of data, plus services for the protection and use of the data. By contrast, a UNIX file provides only persistence. The database system provides protection against corruption due to system or program failure. It also provides concurrency control so that multiple users can share access to data without introducing inconsistencies into the data. Database systems also provide query services to facilitate fast access and manipulation of large data sets. They also provide relationship facilities to model dependencies and relationships among data instances.

An object oriented database management system combines the data management and query capabilities of a traditional database management system with the power and flexibility of the **C++** object oriented programming language.

Before embarking on a discussion of database features, it is important to understand the two different types of data storage: transient and persistent. Transient data exists only while the application program is running and the data variable is in scope. Persistent data exists in long-term storage even after the application stops running.

The basic steps to use a database are as follows.

1. Start a transaction.
2. Create a new database or open an existing database.
3. Read from or write to the database.
4. Close the database.
5. Commit or abort the transaction.

Some of the basic concepts associated with object oriented database systems are defined here.

Transactions: Transactions serve to mark program segments that, from the database point of view, appear to execute all at once or not at all. This feature is important to prevent concurrency anomalies that can arise from the sharing of persistent data. The transaction calls also serve to mark program segments that can be undone. This aspect is important in preventing data corruption due to system or network failure, and also provides the application with a useful undo facility.

Persistent Storage: Persistent storage is the long-term storage of data that exists even after an application stops running. Persistent data objects are created or deleted by using a system supplied overloaded version of the `new` or `delete` operator.

Schema: A schema is a set of class definitions. The database system needs the schema in order to allocate or retrieve objects in persistent memory.

Schema Evolution: Schema evolution refers to the changes undergone by a database schema during the course of the databases existence. A schema evolution facility allows one to redefine the classes in a database schema.

Queries: A query facility is used to select those elements of a set that satisfy a condition specified with a **C++** control expression. For example, one could select only those objects that have `RunNumber` $>$ 2001.

Version Management: Version management facilities allow multiple users to work with multiple versions of an application.

Access Modes: When opening a database, the user can specify read/write or read-only access to the database.

Indices: An index facility marks a data member as an index. This facility instructs the system to maintain an access method which allows efficient retrieval of objects according to the indexed data member. Indices can be ordered (implemented as a B-tree) or unordered (implemented as a hash table). As an example, suppose Calibration is a class with a data member called `RunNumber`. If `RunNumber` is declared to be an index, then Calibration objects can be efficiently located by a query on `RunNumber`. Without an index, a liner search must be used to perform the query.

Associations: Associations are relationships between classes. Associations provide higher level capabilities than simple pointers for modeling and managing relationships between objects. The relationships can be unidirectional or bidirectional, furthermore, they can be one-to-one, one-to-many, many-to-one, or many-to-many.

3 Objectstore

The Objectstore database system contains many features; there are over 100 Objectstore classes and each class has of order 10 member functions. Objectstore is thus very rich in features, but as a consequence, it also has a relatively steep learning curve. Fortunately, only a small subset of the many features are actually needed for a typical application program.

In Objectstore, objects which are to be persistent must first have their class marked in a schema generation file. Persistence is actually initiated by using an overloaded version of the `new` operator. The user must supply a schema generation file, an example of which is listed here for a case consisting of three persistent classes denoted by ClassA, ClassB, and ClassC.

```
#include <ostore/ostore.hh>
#include <ostore/coll.hh>
#include <ostore/manschem.hh>
#include ``ClassA.hh''
#include ``ClassB.hh''
#include ``ClassC.hh''
OS_MARK_SCHEMA_TYPE(ClassA);
OS_MARK_SCHEMA_TYPE(ClassB);
OS_MARK_SCHEMA_TYPE(ClassC);
```

A makefile is used to generate the necessary schema files, which are then compiled and linked with the application source code and Objectstore libraries to create an executable file. The following concepts are specific to Objectstore.

Storage: Objectstore storage has three levels of hierarchy: database, segment, and cluster. A database is a single file and an associated set of processes that provide locking and data access services. (Each database is stored as a separate UNIX file on a disk.) A segment is a set of pages within a database and a cluster is a set of 1 to 15 contiguous pages. Clusters are used to place related objects close to each other in order to achieve efficiency in both disk reads and network transfers.

Collections: A collection is an object that serves to group together other objects. Objectstore has the following subtypes of collections: set, bag, array, and list. A set is an unordered collection that does not permit multiple occurrences of the same object. A bag is an unordered collection that allows multiple occurrences of the same object. Arrays and lists are ordered collections that can either allow or disallow duplicates.

4 Objectivity

In Objectivity, persistence is enabled by using inheritance from a persistent base class. As in Objectstore, persistence is actually initiated by using an overloaded version of the `new` operator. The header files for the persistent classes must be named with the file extension `.ddl` instead of the usual `.h` or `.hh` extension. A makefile is used to process the `.ddl` files, generate the schema files and header files, compile the source code, link the compiled code, and generate an executable file. The following concepts are specific to Objectivity.

Storage: Objectivity uses the following storage hierarchy: federated database (highest level), database, container, and basic object (lowest level). An application opens or creates one federated database which contains a catalog of databases, data type information (schema), and boot file information (lock-sever, host, etc.). The federated database provides file management for the other elements in the hierarchy. Any number of databases may be opened inside the federated database. (The federated database and each individual database are all stored as separate UNIX files on a disk.) At the next level, any number of containers may be opened inside a database (an Objectivity container is roughly equivalent to an Objectstore collection). Finally, basic objects are stored inside the containers. A container may store any number of objects. If a container is not created or specified, the stored object is placed in a default container.

Handles: A handle is a pointer to a persistent object. Handles are used to access the federated database, databases, containers, and basic objects.

5 Benchmark Results

To evaluate and compare the performances of the Objectstore and Objectivity database systems, a simple data model was used to conduct a number of benchmark tests. The tests consisted of measurements of the elapsed time needed to read or write to the database under a variety of conditions. The times quoted in this section always refer to the elapsed time (wall clock time) and were obtained by using the `time` shell command. The disk storage requirements for the two systems were also compared. The results of the benchmark tests are presented in this section along with a brief description of the data model.

5.1 The Data Model

The data model used to test the database systems consisted of two persistent classes: a calibration class containing calibration data and an index class containing ranges of run numbers and times, as well as a pointer to the calibration object which is valid for the specified ranges. The calibration and index objects are stored in separate databases, however, there is a one-to-one correspondence between the calibration and the index objects. The private data members of the two classes are listed here.

```
class Calibration
{
private:
    int   _MyInt[N];     // N = 1, 10, 100, 1000, 10000
    float _MyFloat;
    int   _date;
    int   _time;
    int   _runNumber;
};

class Index
{
private:
    int                       _highDate;
    int                       _highTime;
    int                       _highRunNumber;
    int                       _lowDate;
    int                       _lowTime;
    int                       _lowRunNumber;
    const char*               _dataStoreName;
    os_Reference<Calibration> _target;         // Objectstore reference
};
```

5.2 Write Times

The first set of benchmark tests consist of elapsed write times. Most of the figures in this paper are comprised of two parts: part a) is the result for Objectstore and part b) is the result for Objectivity. Figure 1 shows the elapsed write time as a function of the number of objects written to the database. The five curves correspond to different transaction boundaries and different frequencies of opening and closing the databases. The curves are best explained by indicating where the loop is placed in the main program.

Case 1: (the solid curve with $\diamond$ data points)

```
start transaction
open index database
open calibration database
Begin Loop
    write objects
End Loop
close calibration database
close index database
commit transaction
```

Case 2: (the long-dashed curve with $+$ data points)

```
start transaction
open index database
Begin Loop
    open calibration database
    write objects
    close calibration database
End Loop
close index database
commit transaction
```

Case 3: (the short-dashed curve with $\Box$ data points)

```
start transaction
Begin Loop
    open index database
    open calibration database
    write objects
    close calibration database
    close index database
End Loop
commit transaction
```

Case 4: (the dotted curve with $\times$ data points)

```
Begin Loop
    start transaction
    open index database
    open calibration database
    write objects
    close calibration database
    close index database
    commit transaction
End Loop
```

The solid, long-dashed, and short-dashed curves show the effects of the time overhead associated with opening and closing a database. The time overhead for opening/closing a database is larger for Objectivity. Furthermore, for Objectstore each opened database costs the same amount of time, whereas for Objectivity, the first database takes more time to open than does the second database. (Compare the vertical space between the solid, long-dashed, and short-dashed curves.) The dotted curve illustrates that starting and committing a transaction is a very time consuming task. The dotted curve is the only case in which Objectivity has faster write times. This indicates the Objectivity has a lower transaction overhead. The results so far have been for the time to create a new database and write n objects to it. The dot-dash curve is for the same loop as the solid curve, but now the n objects are written to an existing database. For Objectivity, the write time is reduced by a constant offset of about 4 seconds. For Objectstore, the write time is reduced by about 4 seconds for a small number of objects ($n < 1000$), however, the time reduction becomes negligible for a large number of objects ($n \approx 10000$). The curves in Fig. 1 are for calibration data objects with the data member _MyInt[1].

The effect of the object size on the write time is illustrated in Fig. 2. The calibration object size was varied by changing the size of the _MyInt[N] data member; the solid, long-dashed, short-dashed, and dotted curves are the elapsed write times for $N = 1, 100, 1000$, and 10000, respectively. The program loop is that of Case 1. Objectstore has faster write times for small objects (_MyInt[1] and _MyInt[100]), while Objectivity is faster for large objects (_MyInt[1000] and _MyInt[10000]). The Objectivity results are less sensitive to the object size than are the Objectstore results.

The write speed can be estimated from the slope of the lines in Fig. 2. For the dotted curves (corresponding to _MyInt[10000]), the write speeds are 0.17 sec/object and 0.13 sec/object for Objectstore and Objectivity, respectively. The object size is about 0.04 MB, thus these write speeds correspond to 0.23 MB/sec and 0.31 MB/sec for Objectstore and Objectivity, respectively. These rates should be compared with the raw Unix write speed of 1.1 MB/sec which was obtained by writing objects directly to a Unix file. The above database write speeds correspond to 21% and 28% of the raw Unix write speed for Objectstore and Objectivity, respectively.

5.3 Read Times

The next set of benchmark results are for elapsed read times. The elapsed read time is the time needed to start a transaction, open the index database, find n random objects using a query search, close the database, and commit the transaction. A database was first created by assigning the data members of the i^{th} index object the following values:

$$\begin{aligned} \text{\_lowRunNumber} &= i * 10\,, && (1)\\ \text{\_highRunNumber} &= i * 10 + 9\,, && (2) \end{aligned}$$

where $1 \leq i \leq M$ and is M the total number of objects in the database. Thus the index objects form the sequence (_lowRunNumber, _highRunNumber) = $(10, 19)$, $(20, 29)$, $(30, 39), \ldots, (M, 0)$ where _highRunNumber = 0 in the last object indicates this is the last object in the database. To make the read test, a random run number between 10 and $M + 10$ was generated,

$$\text{runNumber} = x * M + 10\,, \qquad x \in (0, 1)\,, \tag{3}$$

and a query search was done for the index object satisfying

$$\text{runNumber} \geq \text{\_lowRunNumber} \quad \&\& \quad \text{runNumber} \leq \text{\_highRunNumber}\,. \tag{4}$$

Figure 3 shows the elapsed read time as a function of the number of objects read for databases containing 1000 and 10000 objects. For the query search in Eq. (4), the Objectivity read time is significantly faster than the Objectstore read time. The read rate, which includes the search time, can be derived from the slope of the curves. For 1000 objects in the database, the read rates are 64 ms/object for Objectstore and 36 ms/object for Objectivity. Figure 3 also shows that as the number of objects in the database increases, the Objectstore read time degrades much more seriously than does the Objectivity read time. This indicates that Objectivity has better scaling performance. A quantitative measure of the scaling performance can be obtained by forming the ratio of the slopes of the solid to dotted lines:

$$S = \frac{dt/dn\ (1000 \text{ objects in db})}{dt/dn\ (10000 \text{ objects in db})}\,. \tag{5}$$

The scaling factor has the value $S = 1$ when the read rate is independent of the database size. In practice, the read rate decreases as the number of objects in the database increases, thus $0 < S < 1$, with a smaller value of S indicating a poorer scaling performance. The scale factors derived from Fig. 3 are $S = 0.19$ for Objectstore and $S = 0.59$ for Objectivity.

The effect of the calibration object size on the read time is illustrated in Fig. 4. The calibration object size was varied by changing the size of the _MyInt[N] data member. Results are shown for object sizes corresponding to $N = 1, 1000$, and 10000. To a first approximation, the read times should be independent of the calibration object size since the calibration database is not accessed; only the index database is searched and a pointer to the calibration object is returned. Figure 4 shows that the calibration object size has little dependence on the read time. The Objectstore results show only a slight increase in read time with increasing calibration object size. The Objectivity results for _MyInt[1] and _MyInt[1000] exhibit this same behavior, whereas the _MyInt[10000] read times fall slightly below the _MyInt[1] results. This anomalous behavior is probably due to the fact that the _MyInt[1] and _MyInt[1000] results were made when the disk was 97% full and thus heavily fragmented, whereas the _MyInt[10000] results were made after the disk was cleared and only 75% full.

The effect of transactions on the read time are illustrated in Fig. 5. There are 1000 objects in the databases and the calibration objects contain the data-member _MyInt[1]. For the solid curve, a transaction is started, the index database opened, n random objects are queried, the database is closed, and the transaction is committed (this is the same as the solid curves in Figs. 3 and 4). For the long-dashed (short-dashed) curves, a new transaction was started/committed after reading every 10 (100) objects. The read time increases with the frequency of the transactions. Increasing the transaction frequency degrades the Objectstore read times more than it does the Objectivity read times. This indicates that the transaction overhead is more significant for Objectstore that it is for Objectivity. The transaction overhead can be estimated by comparing the times on the curves in Fig. 5 for a fixed number of objects. For example, when the number of objects is $n = 5000$, the data points on the solid, dotted, and dashed curves represent 1, 50, and 500 transactions, respectively. Taking the time difference between the solid and dotted curves at $n = 5000$, and dividing by $50 - 1 = 49$ transactions, yields 0.87 sec/transaction for Objectstore and 0.28 sec/transaction for Objectivity. These numbers are worst case estimates of the cost per transaction.

Figure 6 shows the elapsed time to read one object as a function of the object's position in the database. The object's position is denoted by a real number between 0 and 1, with 0 (1) representing the first (last) object in the database [the object's position is equivalent to x in Eq. (3)]. Results are shown for databases containing 1000 and 10000 objects. The read time includes the time to start a transaction, open the index database, find the object corresponding to a randomly generated run number, close the database, and commit the transaction. This process was repeated 100 times to create the data points shown in the figure. The mean and standard deviation of the 100 read times are given on the figure. Increasing the database size from 1000 to 10000 objects has little effect on the Objectivity read time, but it increases the Objectstore read time by about one second. This once again shows that Objectivity has better scaling behavior in the read mode. A prominent difference between the Objectstore and Objectivity results is that the fluctuations in the read times are much larger for Objectstore then they are for Objectivity.

Figure 7 shows the elapsed read time as a function of the number of databases that are opened, searched, and closed. To make this figure, the calibration class, which was described at the beginning of this section, was cloned into 10 distinct classes (Calibration0, Calibration1, ... etc.) and 10 pairs of databases where created. For the solid curve, the last object in a database consisting of 2 objects was read. For the long-dashed and short-dashed curves, a different random object was queried in each database which contained 1000 objects. For the long-dashed curve, a transaction was started and committed for each database, whereas for the short-dashed curve, a single transaction was used for all the databases. Once again, Objectivity has faster read times than Objectstore, especially for the short-dashed and long-dashed curves which involve query searches. Comparison of the solid curves shows that in the read mode, Objectivity has a smaller overhead for opening/closing databases. Comparison of the short-dashed and long-dashed curves shows that the transaction overhead time is more significant for Objectstore that it is for Objectivity; similar conclusions were drawn from Fig. 5.

The time needed to open and close a database can be estimated from the slope of the curves in Fig. 7. Since the solid curves do not involve any search time (the last object in the database is simply read), they give the best estimate for the database open/close time. The estimated times are 1.17 sec/db for Objectstore and 0.89 sec/db for Objectivity. The one-time overhead per application can be estimated by the first data-point on the solid line. The one-time overhead is 4.62 sec for Objectstore and 4.53 sec for Objectivity.

In Figure 7 a new database was used for each type of calibration object. This storage scheme has a relatively high cost associated with opening each database. An alternative storage scheme would be to put the indices for each type of calibration object into their own container. The elapsed read time verse the number of containers opened is shown in Fig. 8. The advantage of this storage scheme is that the time overhead associated with opening a container is very small, as is evident from the flatness of the curves.

5.4 Disk Storage Requirements

The physical sizes of the Objectstore and Objectivity databases are compared in Fig. 9 where the database size is plotted as a function of the number of objects in the database. Recall that the data model consists of a calibration database and an index database, with a one-to-one correspondence between the objects in each database. Results are shown for calibration objects of various sizes. The calibration object size was varied by changing the number of elements in the data member _MyInt[N]; results are shown for $N = 1, 10, 100$, and 1000. (The size of the index database is independent of the calibration object size.)

The calibration database sizes are very similar for Objectstore and Objectivity when the number of objects is large ($n \approx 10000$). For a small number of objects ($n < 1000$), the Objectivity database is slightly smaller in size, indicating that Objectivity has a lower overhead per object. In contrast, for $n > 2000$ the index database is larger for Objectivity than it is for Objectstore, indicating that Objectivity has a higher overhead associated with its indices. It should also be noted that Objectivity has a federated database which is about 1 MB in size for this data model. The federated database contains a catalog of databases, data type information (schema), and boot file information (lock-sever, host, etc.).

6 Conclusions and Recommendations

The feasibility of using object oriented database management systems to store and retrieve conditions information for the BaBar experiment has been examined. Two commercial object oriented database management systems, Objectstore and Objectivity, have been evaluated and compared. The writing, reading, and disk storage performance of the two systems has been measured in a series of benchmark tests. The goal of this study is to make a recommendation for the choice of the database system to use in the BaBar experiment. A summary of the most important performance characteristics for the two database systems is given in the following outline.

Disk Storage Requirements: The disk storage requirements of the two database systems are very similar and thus disk storage performance is not a factor for selecting a database system.

Writing: Objectstore has a faster write speed for small objects while Objectivity is faster for large objects. In the write mode, Objectivity has a higher overhead for opening/closing a database, whereas Objectstore has a higher overhead for starting/committing a transaction. The the write performance is very dependent on the data model and on the software design. Fortunately, writing is a one-time operation and the write performance is not a crucial factor in choosing a database system. Since neither system exhibits a clear superiority in writing performance, either system would be acceptable for use in the BaBar experiment.

Reading: In all of the read-search benchmark tests, Objectivity clearly out performs Objectstore: Objectivity has faster read-search times and exhibits better scaling behavior. Furthermore, in the read mode, Objectivity also has smaller overheads for starting/committing transactions and opening/closing databases. In off-line analysis, the read-search performance of the database is the most important feature to consider when selecting a database system.

A summary of the most important performance measurements for Objectstore and Objectivity is given in Table 1. In general, Objectivity exhibits better performance in all of the measured categories, and especially in the critical area of read-search performance. Based on its superior read-search performance and scaling performance, Objectivity is clearly the better database system for use in the BaBar experiment. We therefore recommend that the BaBar collaboration adopt the Objectivity database system for the storage and retrieval of conditions information.

In addition to the performance results already cited, there are other non-technical reasons for selecting Objectivity for use in the BaBar experiment. First of all, Objectivity has been used extensively by the CERN RD46 collaboration. This group has been doing research and development work on object oriented software technology for use in high energy physics experiments. BaBar could benefit from the experience and published results of the RD46 collaboration. Finally, Objectivity has been receptive to granting educational discounts on licenses for their software. This is an important consideration since the BaBar computing budget is very limited.

Now that a database system has been selected, the next step is to develop prototype code for storing and retrieving conditions information for the BaBar experiment. The goal is to make an interface such that the database system calls are invisible to the user. This is important since it will relieve the user from the time consuming task of learning how to use the database system. Development of the prototype code is currently in progress.

References

[1] Objectstore **C++** Application Programming Interface User Guide, Object Design, Inc., Burlington, MA, 1995.

[2] Objectivity/DB **C++** Developer, Objectivity, Inc., Mountain View, CA, 1995.

Quantity	Objectstore	Objectivity
One-time overhead per application	4.6 sec	4.5 sec
Cost per transaction	0.87 sec	0.28 sec
Cost per db open/close	1.17 sec	0.89 sec
db write/Unix write	0.21	0.28
Read-search access performance	64 ms/object	36 ms/object
Scaling performance	0.19	0.59

Table 1: A summary of performance measurements for Objectstore and Objectivity.

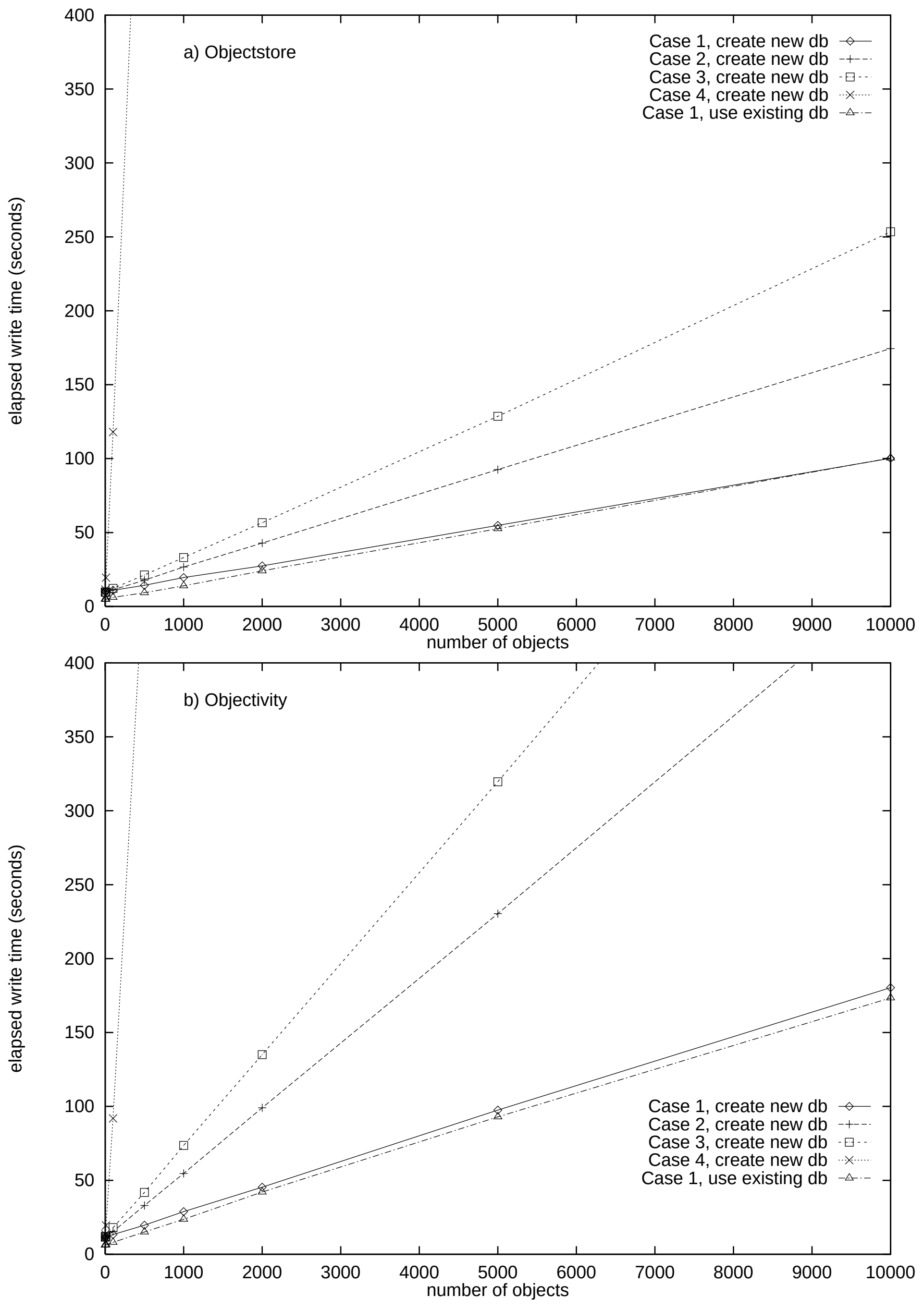
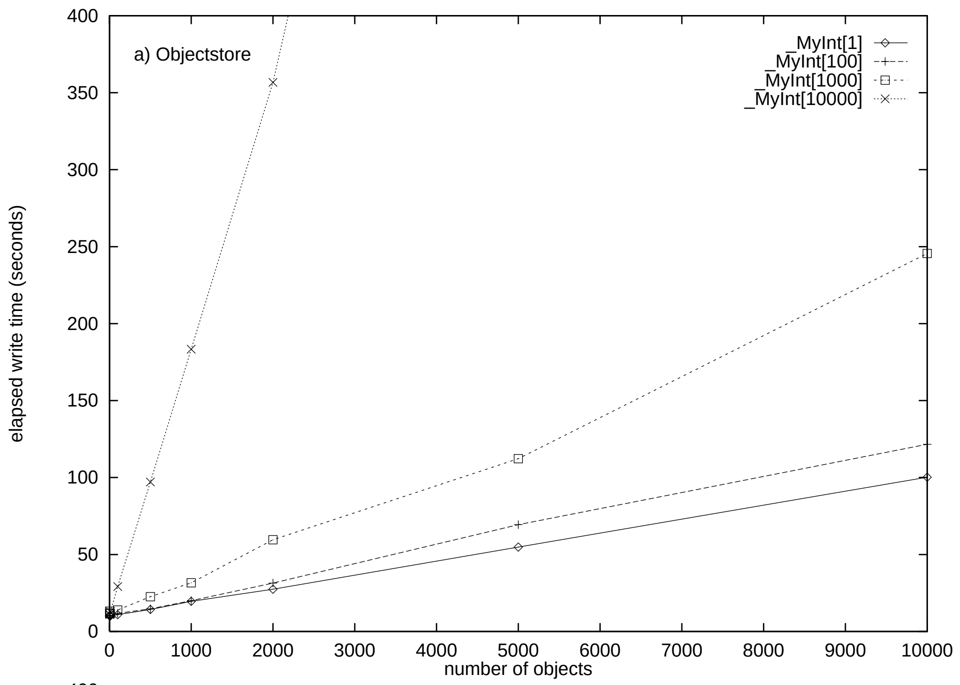


Figure 1: Elapsed write time as a function of the number of objects written to the database for various transaction scenarios. The solid, long-dashed, short-dashed, and dotted curves correspond to transaction cases 1, 2, 3, and 4, respectively, which are described in the text. These write times include the time to create a new database. The dot-dashed curve is also for transaction case 1, but the objects are written to an existing database. Parts a) and b) are for Objectstore and Objectivity, respectively.



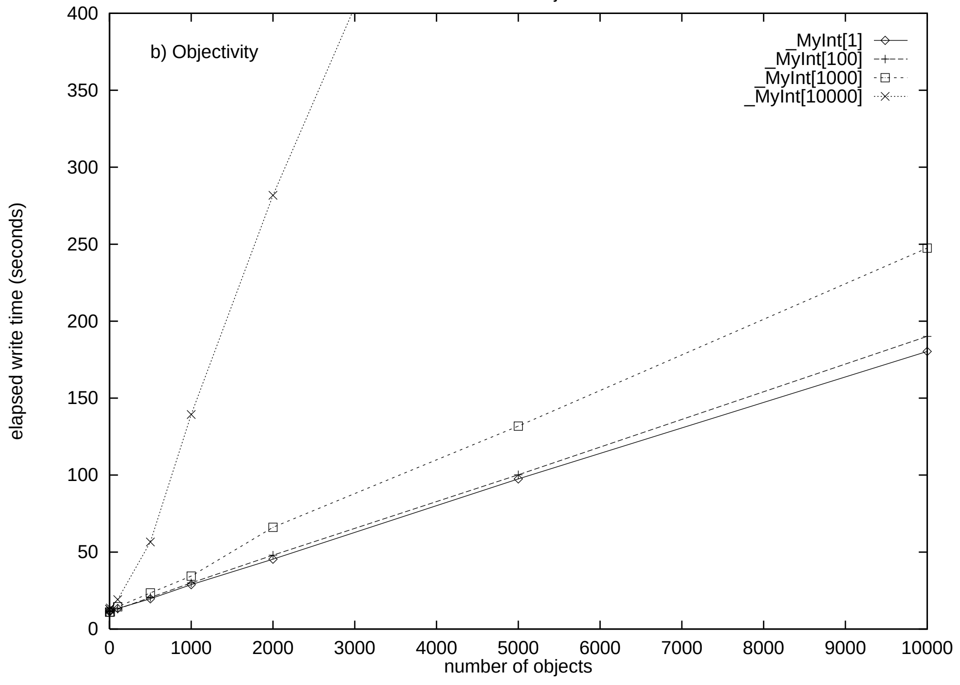


Figure 2: Elapsed write time as a function of the number of objects written to the database for objects of various sizes. The object size is varied by changing the size of the _MyInt[N] data member. The solid, long-dashed, short-dashed, and dotted curves are for object sizes corresponding to $N = 1, 100, 1000$, and 10000, respectively. Parts a) and b) are for Objectstore and Objectivity, respectively.

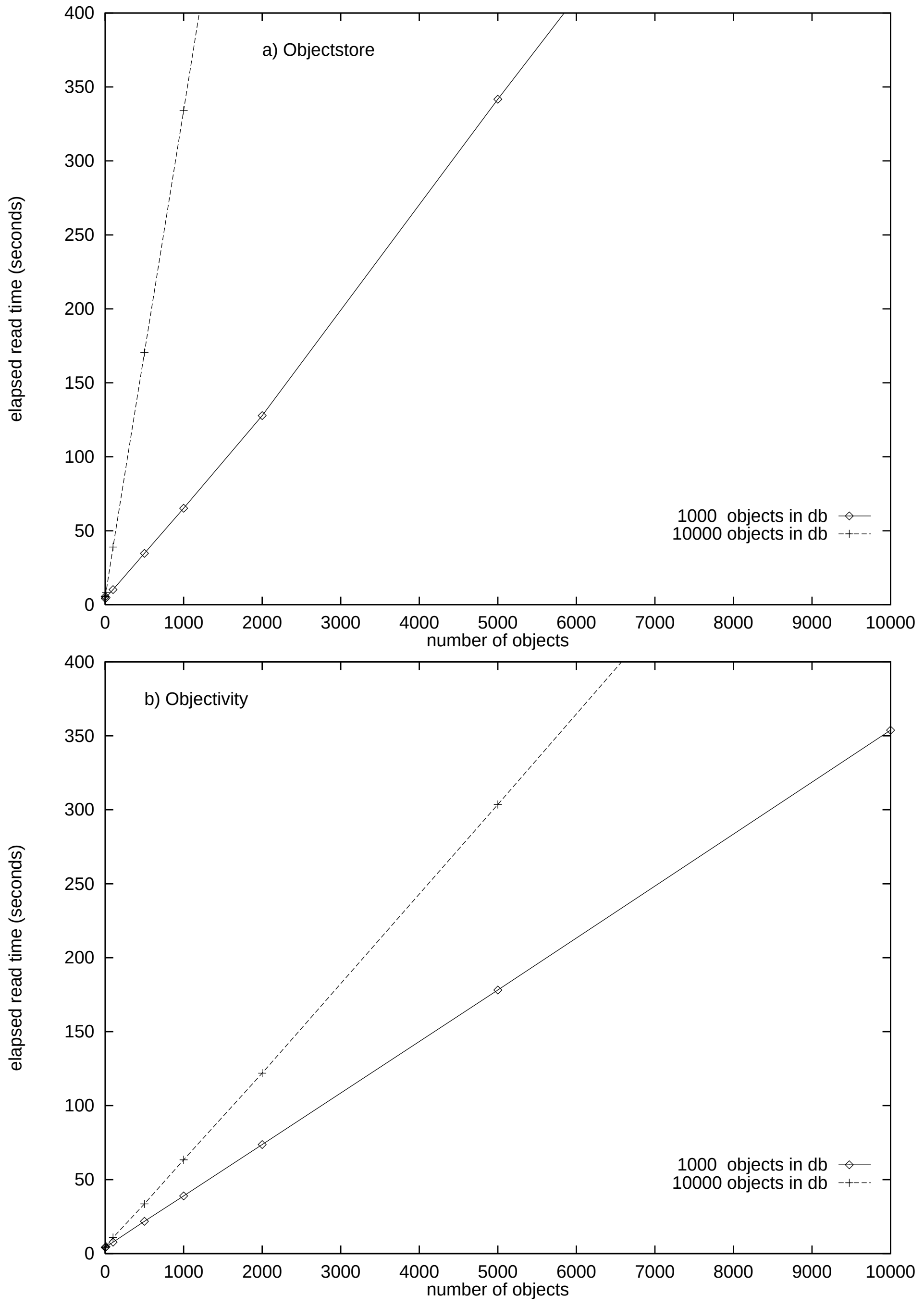
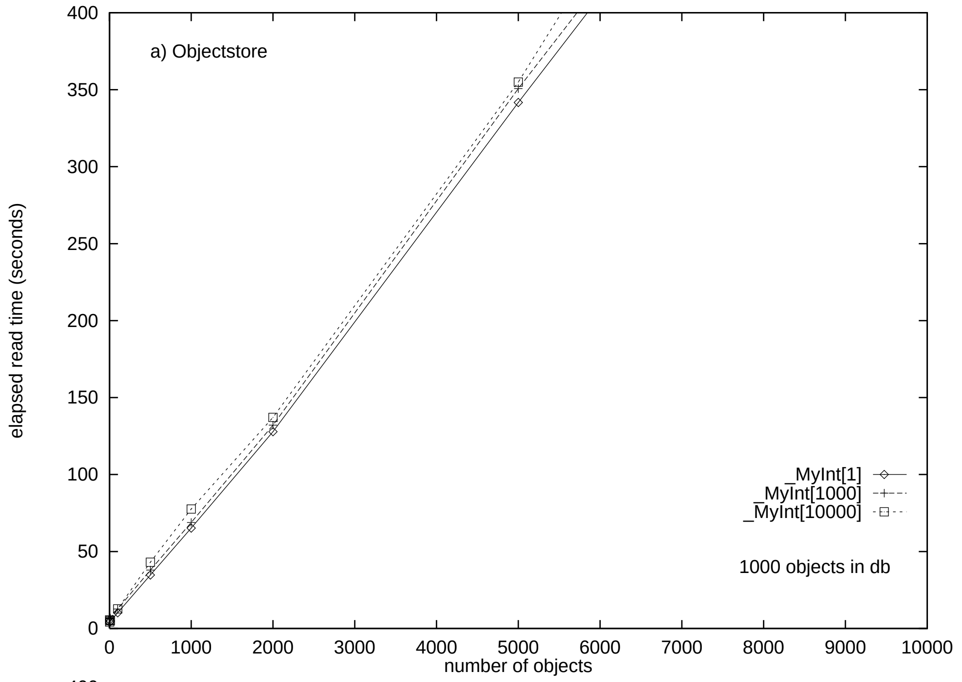


Figure 3: Elapsed read time as a function of the number of objects read from the database. The solid (dashed) line is for a database containing 1000 (10000) objects. Parts a) and b) are for Objectstore and Objectivity, respectively.



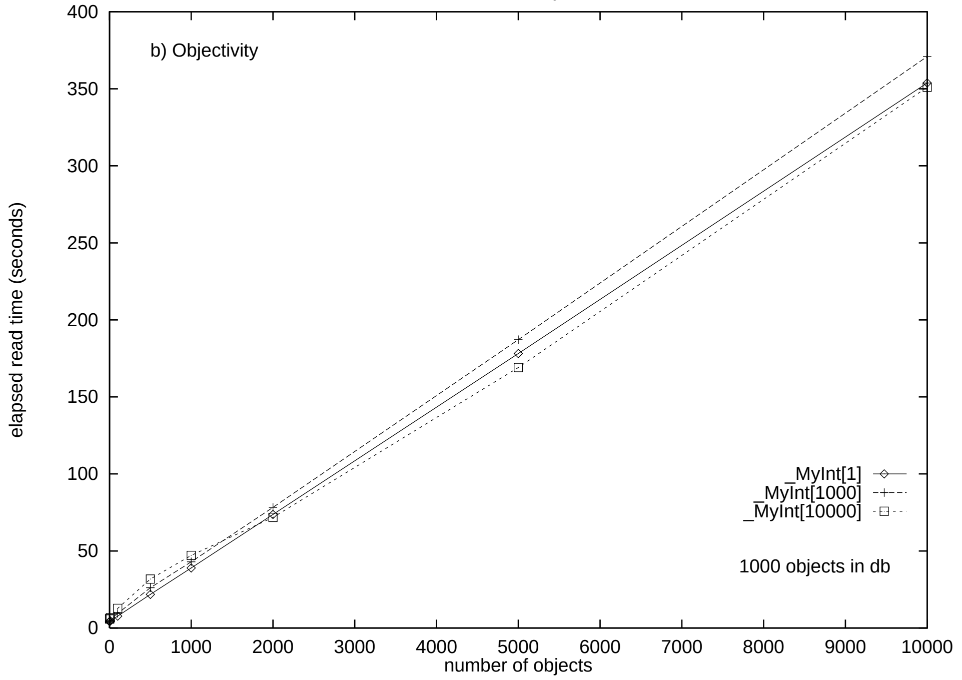


Figure 4: Elapsed read time as a function of the number of objects read from the database for objects of various sizes. The object size is varied by changing the size of the _MyInt[N] data member. The solid, long-dashed, and short-dashed curves are for object sizes corresponding to $N = 1, 1000$, and 10000, respectively. Parts a) and b) are for Objectstore and Objectivity, respectively.

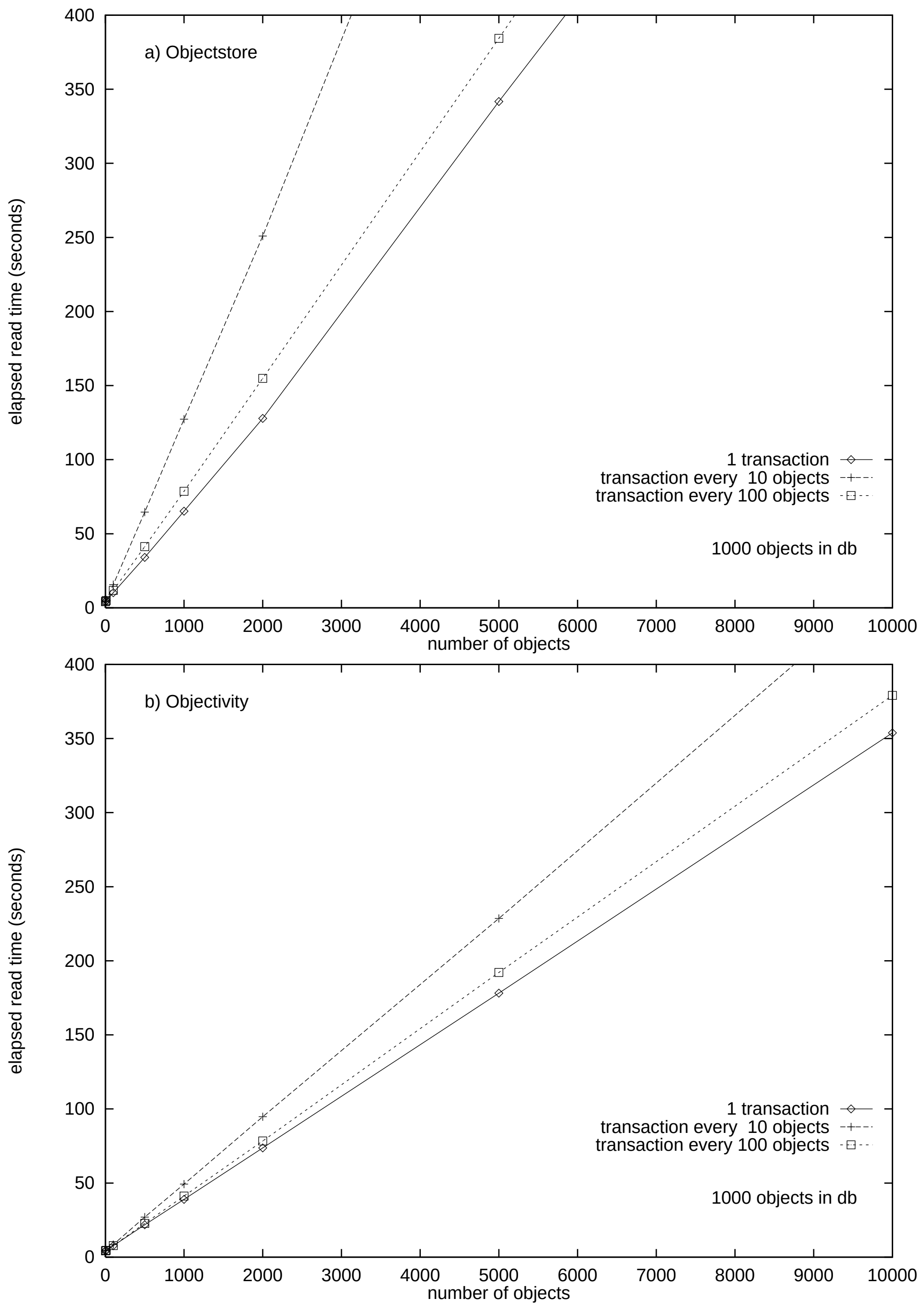


Figure 5: Elapsed read time as a function of the number of objects read from the database for various transaction intervals. For the solid line, one transaction was used for the entire read process. For the long-dashed (short-dashed) line, a new transaction was started after reading every 10 (100) objects. Parts a) and b) are for Objectstore and Objectivity, respectively.

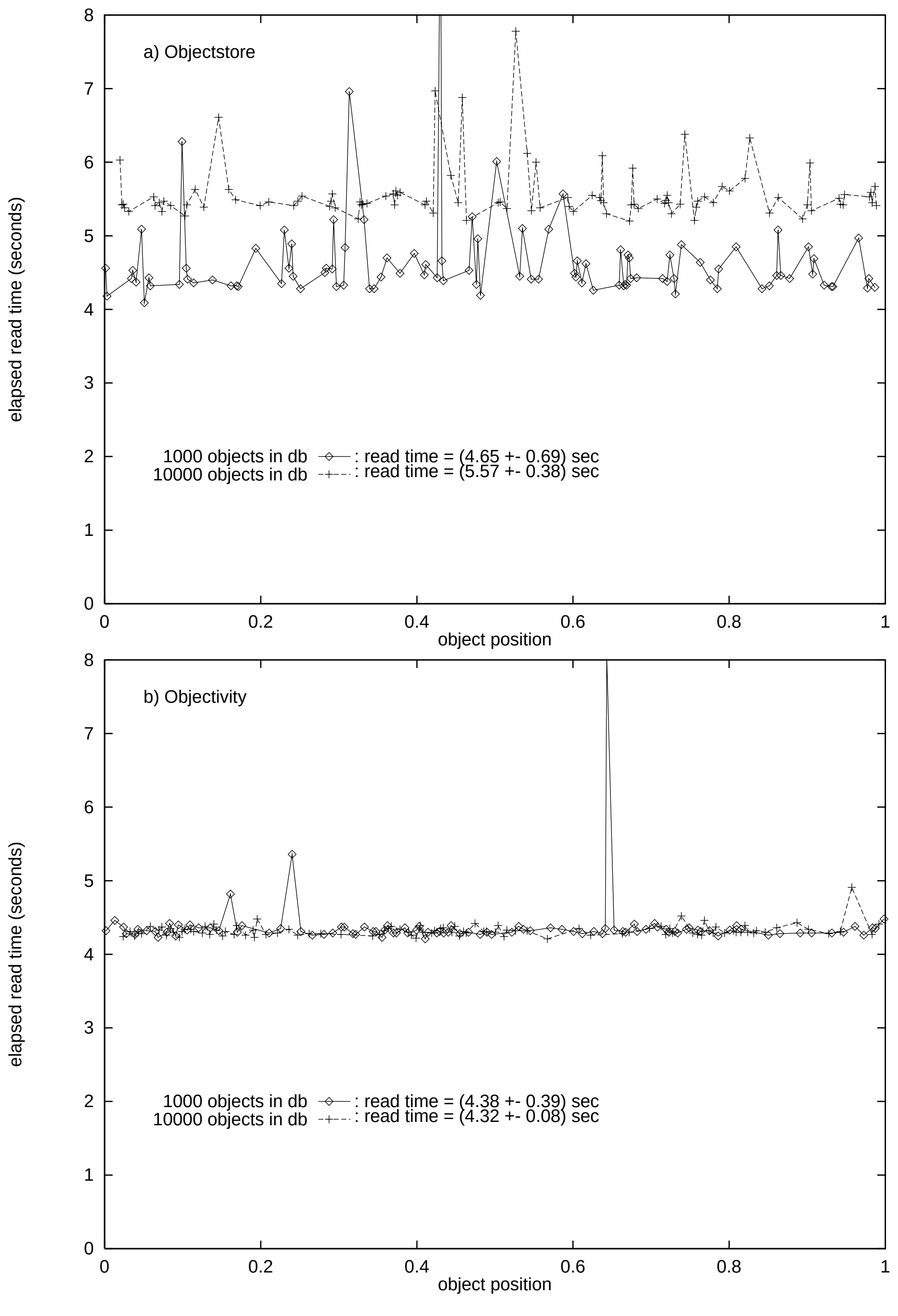


Figure 6: Elapsed time to read one object as a function of the object's position in the database. The solid (dashed) curve is for a database containing 1000 (10000) objects. Parts a) and b) are for Objectstore and Objectivity, respectively. The spikes which go off the scale are not reproduceable.

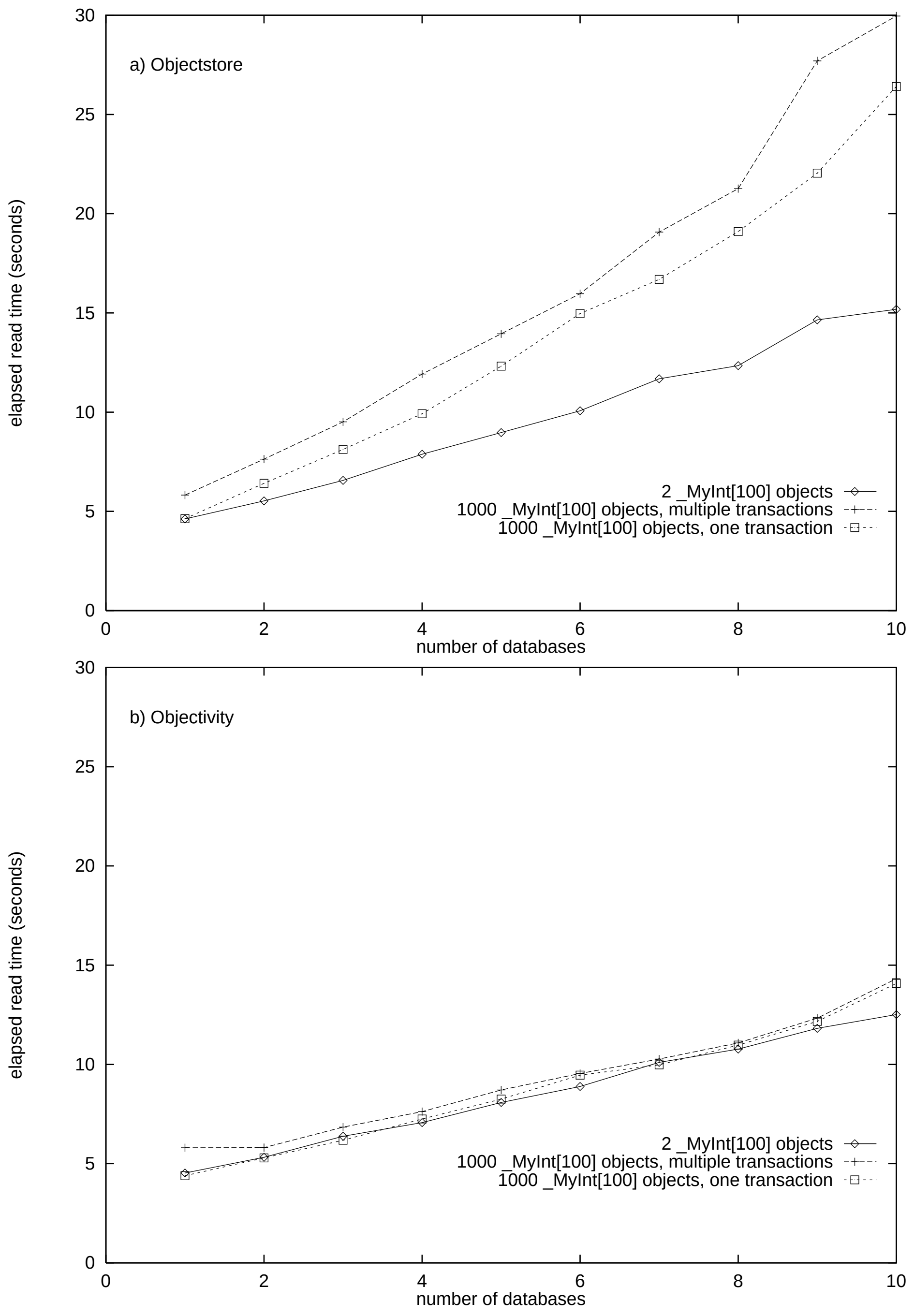


Figure 7: Elapsed read time as a function of the number of opened databases. A single object was read from each opened database. For the solid curve, the last element of a database consisting of 2 objects was read. For the dashed curves, a random object was queried from a database consisting of 1000 objects. For the short-dashed curve, a single transaction was used to read all the databases, whereas for the long-dashed curve, separate transactions were used to read each database. Parts a) and b) are for Objectstore and Objectivity, respectively.

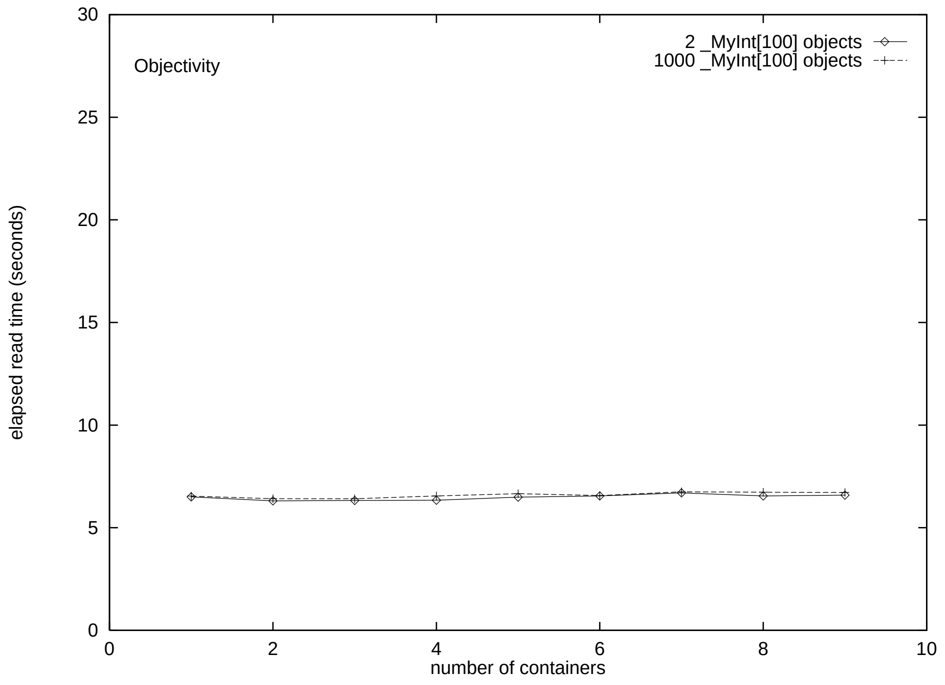
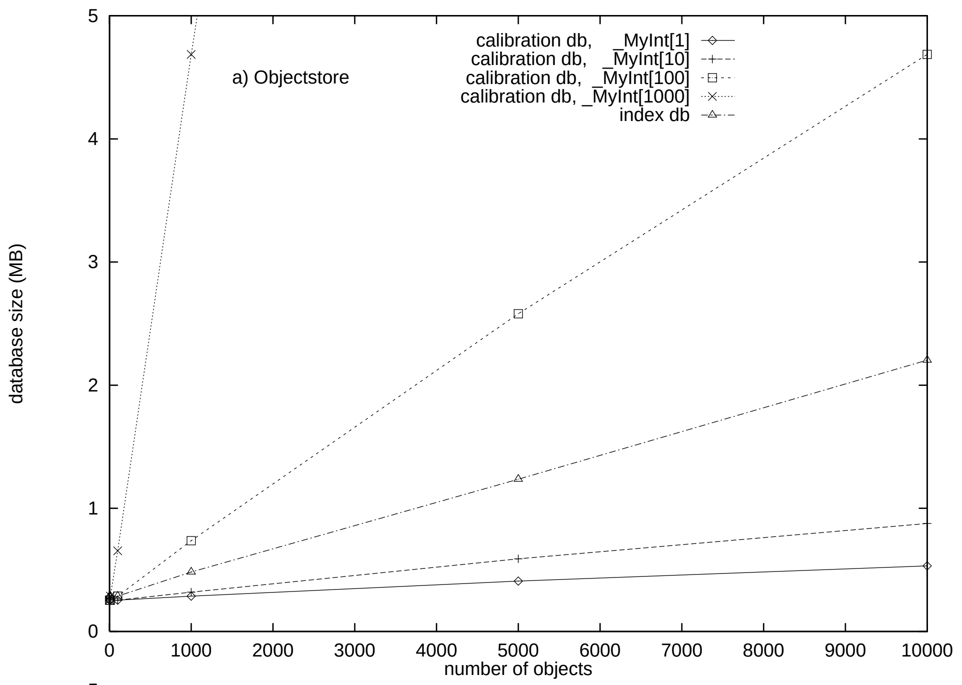


Figure 8: Elapsed read time as a function of the number of opened containers. A single object was read from each opened container. For the solid curve, the last element of a database consisting of 2 objects was read. For the dashed curve, a random object was queried from a container consisting of 1000 objects. A single transaction was used to read all the containers.



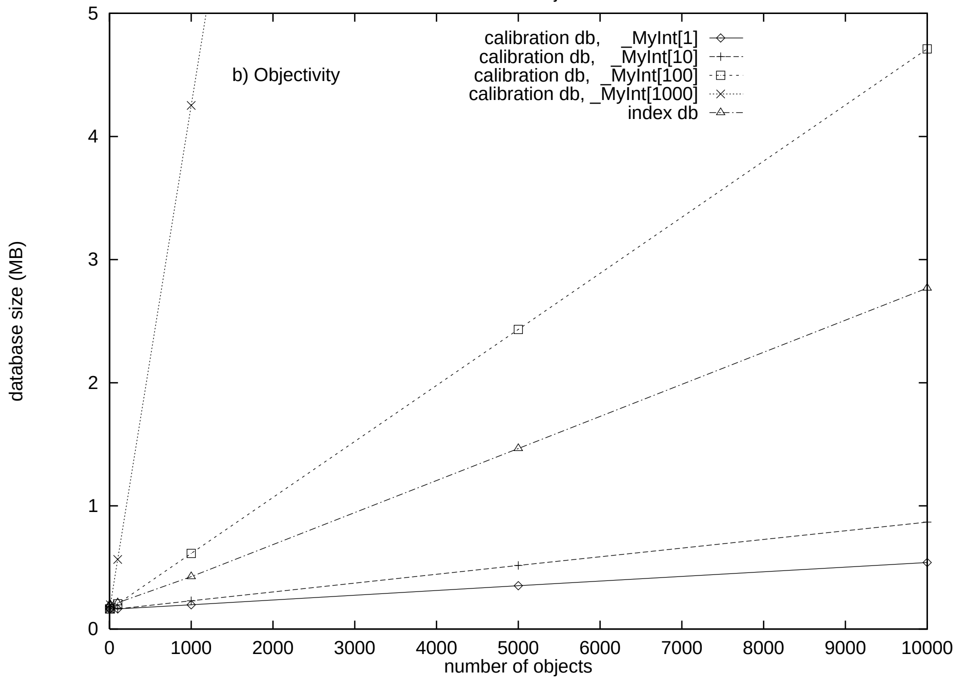


Figure 9: Size of the database as a function of the number of objects in the database for objects of various sizes. The calibration object size is varied by changing the size of the _MyInt[N] data member. The solid, long-dashed, short-dashed, and dotted curves are for calibration object sizes corresponding to $N = 1, 10, 100$, and 1000, respectively. The dot-dashed curve is for the size of the index database. Parts a) and b) are for Objectstore and Objectivity, respectively.